COMSC-210 Lecture Topic 10
Hash Tables

Reference
Childs Ch. 11

O(n) Lookups...
AssociativeArray's lookups are O(n)
StaticArray and DynamicArray are O(1)

How To Maintain O(1)?
need to avoid O(n) lookup, to match the key
one solution: convert key to an index!

"hashing" -- converting a key into a number
a *single* whole number, +/-
`int hashCode(const string& x) {return x.size();}`
...or better, sum of ascii codes for *all* chars

The "Hash Code"
it's a key converted to integer form (above)
(yes, it's possible for different keys to yield the same hash code value)
it's the default *array* index for the object
so that it can readily find itself
in any array by jumping *directly*
to the element where it "wants" to be
value range: +/- 2 billion (i.e., the signed int range)

Practical Problems Of Hashing
unordered
there has to be lots of empty space in the array
to accommodate the possible range of hash codes
"collisions" are possible -- different keys whose
hash code value is the same!
the #of unique hash codes may exceed the #of
elements in the array (capacity)
"holes" in the array
no used/unused separation
track size AND capacity
duplicate keys *not* okay
(but value can be a list of values)

The "Wrapped Index"
how to fit into an array whose size does not
span the range of hash code values

`wrapped index = hash code % array capacity`

always in the range 0 to cap-1?
except for negative hash codes
so `if (wrapped index < 0)`
`wrapped index += array capacity`
a handy private function: `getWrappedIndex(const U&)`
get hash code by calling the hashCode function
modulus with array capacity
if negative, add array capacity
return the `int` result
increases possibility of collisions...

Getting The hashCode To The HashTable
HashTable is generic -- works for any data type
hashCode is specific to the data type used in the application
so function gets written in `main`
to share with HashTable object, use *pointer to function*

Load Factors And Array Expansion
if using self-adjusting dynamic arrays
array needs to remain "sparse", to minimize collisions
solution: array expansion when array becomes "too full"
detecting "too full": calculate "load factor"
#of used elements / array capacity
if load factor exceeds maximum allowable load factor,
double the array capacity

BUT do not use simple array copy, because
wrapped indexes are function of array capacity *(rehashing)*

Handling Collisions
still, collisions will occur -- possible design solutions:
overflowing into unused adjacent elements (**probing**)
stacking data in a single array element (**chaining**)

Linear Probing v.1.0
for illustration only -- do not use as is
use unused adjacent index if wrapped index in use
"linear probing" -- traverse array from wrapped index
1. get wrapped index (w.i.)
2. for-loop to look at ALL elements, starting at w.i.
if inUse and key matches, return value
if not inUse, save, set inUse, return value

Linear Probing v.2.0
BUT what about deleted keys?
do not insert at unused position if
a duplicate key exists someplace else
need to traverse *entire array* just in case! -- *O(n)*

Linear Probing v.2.1
a better way -- back to *almost* O(1)
no reason to traverse past a location if
nothing was ever stored there
because "touching" would use that location
before using any after that one
need a way to distinguish bwtn *never*-used and *previously*-used
then traversals can stop at never-used
replace `bool* inUse` with `int* status` array
0 = "in use"
1 = "never in use"
2 = "no longer in use"
modify for-loop:
2. for-loop to look at ALL elements, starting at w.i.
if inUse and key matches, return value
if **"never in use"**, save, set inUse, return value
requires "rehashing" from time to time...

Chaining
use array of STL `list`s, to stack data at each index
"jagged" rows
"inUse" not tracked: use `list` at w.i.
sequential search of `list` makes for approx. O(1)

using static array (to simply coding...)
change the data members:

```
template <class T, class U, int CAPACITY>
```

use constructor parameter to share function's location:

```
HashTable<int, string> phoneBook(hashCode); // in main

HashTable(int(*)(const U&)); // constructor prototype
int(*hashCode)(const U&); // as data member, "hashCode"

// in getWrappedIndex:
  int w.i. = hashCode(key) % cap;
  if (w.i. < 0) w.i. += cap;
  return w.i.;
```

`operator[]` Setter

1. get "wrapped index", 0 to cap-1
2. if inUse and key matches, return value
3. if not in use, ++siz, and...
   save key at index
   set inUse to true at index
   return unset value at index
4. if else COLLISION -- used by a different key!

`operator[]` Getter

1. get "wrapped index", 0 to cap-1
2. if inUse and key matches, return value
3. return dummy

Avoiding Collisions

collisions will happen, no matter what
  but...
to make it less likely for separate hash codes
  to result in same wrapped index
1. let array *capacity* be a prime number
2. judicious hash code calculations

```
class HashTable
{
  struct Node
  {
    T value;
    U key;
  };

  list<Node> data[CAPACITY];
  ...
```

redefine "capacity"...

`int capacity() const {return 0.8 * CAPACITY;}` // rule-of-thumb
possible to go over capacity, but big oh deteriorates

operator[ ] setter (using the STL algorithm `find`):

```
w.i. = (wrapped index to store data item)
typename list<Node>::iterator it;
Node temp; temp.key = key; // key is parameter
it = find(data[w.i.].begin(), data[w.i.].end(), temp);
if (it == data[w.i.].end()) // no matching key
{
  increment siz
  data[w.i.].push_back(temp)
  return data[w.i.].back().value
}
else return it->value
```

to support STL find, add this to Node

```
void operator=(const T& v){value = v;}
bool operator==(const Node& n) const {return key == n.key;}
```

Chaining With A Static Array Template

`HashTable<int, string, 1009> phoneBook(hashCode);` // in main

About the STL list Template

Libraries:

```
#include <list> // for the list itself
#include <algorithm> // for searching the list
using namespace std;
```

Variables:

```
list<Node> l; // an empty list of Node objects [FYI only -- it's just one list]
list<Node> data[N]; // an array of N empty lists [a data member]
typename list<Node>::iterator it; // an uninitialized mutating pointer to a Node in a list [a local variable]
typename list<Node>::const_iterator it; // an uninitialized read-only pointer to a Node in a list [a local variable]
int wi; // the wrapped index, in the range 0 to N-1 [a local variable]

C++11 code:
auto it = data[wi].begin(); // a mutating "it" [a local variable]
auto it = data[wi].cbegin(); // a read-only "it" [a local variable]
...replace begin() and end() with cbegin() and cend() when using read-only "it"

data[wi] -- the list at the wrapped index "wi"
data[wi].size() -- how many Node objects are in the list
```

Adding 1 Node named "node" to the list at wrapped index, wi

```
Node temp;
temp.key = key; // if any (usually so), of type U
temp.value = value; // if any (usually not), of type T
data[wi].push_back(node);
```

Finding a Node with a matching key named "key":

```
Node temp; temp.key = key; // do NOT set temp.value
```

```
it = find(data[wi].begin(), data[wi].end(), temp);
if (it == data[wi].end())
  not found
else
{
  it->value -- is the value at the found Node
}
```

Deleting a Node pointed to by "it" in the list at the wrapped index "wi":

```
list[wi].erase(it);
```

Deleting ALL Nodes in the list at the wrapped index "wi":

```
list[wi].clear();
```

Loop through all lists in an array of lists, "data":

```
for (wi = 0; wi < N; wi++)
  data[wi] -- the list at wrapped index "wi"
```

Loop through all Node objects in the list at wrapped index "wi":

```
for (it = data[wi].begin(); it != data[wi].end(); it++)
  it->key the Node's key
  it->value the Node's value
```